本文档将会把整个 Geatpy 工具箱的使用方法再次捋一下,把关键点、核心点梳理 以便,以达到"理解本质、化繁为简"的目的。

## Geatpy 工具箱由:内核函数 以及进化算法框架(简称 EA 框架)两部分组成。

s.t. -1 <= x <= 2

如:

一、Geatpy 工具箱的组成及基本用法

内核函数是封装起来的高性能计算单元,内部有数万行代码,它与 EA 框架是高度

脱耦合的,可以在 python 上直接调用这些内核函数。内核函数一览可见网站: https://github.com/geatpy-dev/geatpy/tree/master/ core/Windows/lib64/v3.5 以变异函数"mutpolyn"为例,其具体用法可通过 import geatpy as ea; help(ea.mutpolyn)

查看。

EA 框架由:种群类、问题类、算法模板类、进化算子类四个大类组成。这里如果对 类的概念不太熟悉,可先通过互联网学习一下 Python 的面向对象编程的相关知识。以 soea\_demol 为例, 在使用 Geatpy 解决一个具体的优化问题时, 首先要做的是编写一个自

定义问题类,然后编写一个执行脚本(在案例中通常被命名为 main.py)来启动 Geatpy。 其中最核心的是编写自定义问题类,需要根据具体的优化模型,确定好决策变量、待优 化的目标函数以及约束条件来编写具体的代码。例如: 🗖 main. py 🗵 MyProblem. py 🗵 # -\*- coding: utf-8 import numpy as np import geatpy as ea 该案例展示了一个简单的连续型决策变量最大化目标的单目标优化问题。  $\max f = x * np.sin(10 * np.pi * x) + 2.0$ 

```
10 ▼ class MyProblem(ea.Problem): # 維承Problem父类
  11 🔻
          def __init__(self):
             name = 'MyProblem' # 初始化name (函数名称,可以随意设置)
  12
             M = 1 # 初始化M (目标维数)
  13
             maxormins = [-1] # 初始化maxormins (目标最小最大化标记列表, 1: 最小化该目标; -1: 最大化该目标)
             Dim = 1 # 初始化Dim (決策变量维数)
  15
             varTypes = [0] * Dim # 初始化varTypes (决策变量的类型,元素为0表示对应的变量是连续的; 1表示是离散的)
  16
  17
             1b = [-1] # 決策变量下界
  18
             ub = [2] # 決策变量上界
             lbin = [1] * Dim # 决策变量下边界 (0表示不包含该变量的下边界, 1表示包含)
  19
  20
             ubin = [1] * Dim # 決策变量上边界 (0表示不包含该变量的上边界, 1表示包含)
  21
             # 调用父类构造方法完成实例化
  22
             ea.Problem.__init__(self, name, M, maxormins, Dim, varTypes, lb, ub, lbin, ubin)
  23
          def aimFunc(self, pop): # 月标函数
  24 🔻
  25
             x = pop.Phen # 得到決策变量矩阵
  26
             pop.0bjV = x * np.sin(10 * np.pi * x) + 2.0 # 计算目标函数值,赋值给pop种群对象的ObjV属性
                                           图 1
为 3,则 [0]*Dim 相当于 [0,0,0]。
```

由于 Geatpy 的数据结构规定种群的表现型矩阵 Phen(一般等价于代表模型中的决

策变量)是 Numpy ndarray 类型的矩阵,这意味着 aimFunc()中假如无法通过矩阵运算的

方法"同时"计算种群所有个体的目标函数值,那么需要用循环或者并行并发、分布式

等方法分别计算种群每个个体的目标函数值,最后拼成一个种群染色体矩阵 ObjV。例

 $Phen = \left(\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 1 & 2 & 3 \\ 3 & 1 & 2 & 3 \end{array}\right)$ 那么利用 Phen[0,:] 便可以得到种群的第一个个体的"决策变量"。以此类推。 注意:Geatpy 中"决策变量"的含义仅仅指的是种群染色体矩阵 Chrom 根据其编

码设定("RI", "P"或"BG")进行一次解码后得到的内容,比如假设编码为"BG"

的染色体: 111011 解码后为: 73, 则这个"73"便被称作决策变量。假如用户需要

• 实例化问题类的对象; • 实例化一个不带染色体信息的种群对象;

▼ if \_\_name\_\_ == '\_\_main\_\_': -----实例化问题对象 problem = MyProblem() # 生成问题对象

NIND = 40 # 种群规模

• 实例化算法模板对象;

称作是"决策变量"。

工作是:

Field = ea.crtfld(Encoding, problem.varTypes, problem.ranges, problem.borders) # 创建区域描述器 population = ea.Population(Encoding, Field, NIND) # 实例化种群对象(此时种群还没被初始化,仅仅是完成种群对象的实例化) 12 13 -----算法参数设置-

BestIndi.save() # 把最优个体的信息保存到。

print('最优的控制变量值为:')

print('没找到可行解。')

for i in range(BestIndi.Phen.shape[1]): print(BestIndi.Phen[0, i])

---种群设置--

-----输出结果:

[BestIndi, population] = myAlgorithm.run() # 执行算法模板,得到最优个体以及最后一代种群

• 调用算法模板对象的 run() 函数进行进化优化;

14 myAlgorithm = ea.soea\_SEGA\_templet(problem, population) # 实例化一个算法模板对象 myAlgorithm.MAXGEN = 25 # 最大进化代数 myAlgorithm.logTras = 1 # 设置每隔多少代记录日志,若设置成6则表示不记录日志 myAlgorithm.verbose = True # 设置是否打印输出日志信息 myAlgorithm.drawing = 1 # 设置绘图方式(6.不绘图,1:绘制结果图,2:绘制目标空间过程动画;3:绘制决策空间过程动画) 15 16 17

22 23 print('评价次数: %s' % myAlgorithm.evalsNum) print('时间已过 %s 秒' % myAlgorithm.passTime) 24 if BestIndi.sizes != 0: print('最优的目标函数值为: %s' % BestIndi.ObjV[0][0])

10

18 19

20 21

27 28 29

31

整数)。

rithm.py 文件。

数值矩阵 ObjV 来计算得出。

算法模板 soea\_SEGA\_templet 的内容如下图:

```
图 2
  这里需要强调的是种群染色体的编码方式。Geatpy 内置 3 种基本的种群染色体编
码:实整数编码 "RI"、排列编码 "P"以及二进制/格雷码编码 "BG"。前两个编码的种
群的染色体矩阵 Chrom 和种群表现型矩阵 Phen 的内容是一模一样的(详见《Geatpy 数
据结构》章节); 而"BG"编码的种群染色体矩阵 Chrom 是一个元素全为 0、1 的矩阵,
它要通过解码才能得到种群表现型矩阵 Phen。
  二、Geatpy 数据流动情况简述
  本节将描述 Geatpy 中的数据是怎么流动的,您可以通过单步调试的方法对照着本
文进行学习,以图 2 为例,完整代码在 soea demo1 中。
  1、从执行脚本 main.py 开始看,实例化问题类对象的过程就不用多说了,首先发
生重要的数据流动的是创建区域描述器 Field。它是调用 ea.crtfld() 这个内核函数完成的,
具体 API 可通过 import geatpy as ea; help(ea.crtfld) 查看。区域描述器的概念可见《Geatpy
数据结构》章节,它并非很特殊的数据结构,而仅仅是一个 Numpy ndarray 类型的矩阵,
用于描述与种群染色体有关的"范围信息"。比如实整数编码的染色体对应的区域描述
器是一个 3 行 N 列的矩阵,第一行代表染色体每一位数字的下界,第二行代表染色体
```

每一位数字的上界,第三行代表染色体每一位数字的"类型"(0表示是实数;1表示是

化,仅仅是完成种群对象的实例化"。一般来说种群对象的染色体矩阵 Chrom 被赋值之

因此将会跳转到 soea\_SEGA\_templet.py 文件的构造函数中。如下图所示:

3、接着是实例化算法模板对象。这里由于要实例化的类为"soea\_SEGA\_templet",

2、然后到实例化种群对象,这里将会跳转到 Population.py 文件中,执行种群类的

 $\operatorname{def}$  \_\_init\_\_(self, problem, population): ea.SoeaAlgorithm.\_\_init\_\_(self, problem, population) # 先週用父类构造方法 if population.ChromNum != 1: raise RuntimeError('传入的种群对象必须是单染色体的种群类型。') self.name = 'SEGA' self.selFunc = 'tour' # 錦标賽选择算子 if population.Encoding == 'P': self.recOper = ea.Xovpmx(XOVR = 0.7) # 生成部分匹配交叉算子对象 self.mutOper = ea.Mutinv(Pm = 0.5) # 生成逆转变异算子对象 self.recOper = ea.Xovdp(XOVR = 0.7) # 生成两点交叉算子对象 elif population.Encoding == 'RI': self.mutOper = ea.Mutbga(Pm = 1/self.problem.Dim, MutShrink = 0.5, Gradient = 20) # 生成breeder GA变异算子对象 raise RuntimeError('编码方式必须为''BG''、''RI''或''P''.') 图 3 可见这里根据种群染色体的编码类型而实例化了合适的算法对象。比如实例化一个

为让两个种群合并为一个大的种群,中间过程是两个种群类的各个属性(如染色体矩阵 Chrom、目标函数值矩阵 ObjV 等等)的合并。 在进化优化中,一定要注意"种群的适应度"和"种群的目标函数值"是不一样的,

前者是一个列向量,后者是一个矩阵。如果进化算法的流程里需要依靠"适应度"来选

择个体,那么需要给种群对象的 FitnV 属性赋值。一般来说适应度可以根据种群目标函

elf.selFunc = 'tour' # 锦标赛选择算子 self.recOper = ea.Xovdp(XOVR = 0.7) # 生成两点交叉算子对象 if population.Encoding == 'BG':
 self.mutOper = ea.Mutbin(Pm = None) # 生成二进制变异等子对象,Pm设置为None时,具体数值取变异等子中Pm的默认值 elif population.Encoding == 'RI':

self.mutOper = ea.Mutbga(Pm = 1/self.problem.Dim, MutShrink = 0.5, Gradient = 20) # 生成breeder GA变异寡子对象 else: raise RuntimeError('编码方式必须为''BG''、''RI''或''P''.')

population = population[ea.selecting('dup', population.FitnV, NIND)] # 采用基于适应度排序的直接复制选择生成新一代种新 return self.finishing(population) # 调用finishing完成后续工作并返回结果 图 4

6、执行完算法模板对象的 run() 函数后,返回最优个体以及最后一代种群。 以上便是使用 Geatpy 进行进化优化的数据流动的情况简述。可以看到 Geatpy 的 EA 框架的基于 Geatpy 的工具箱内核函数来实现运作的,可以明显感受到模块之间的耦

合程度比较小,可以很清晰地看到进化算法的整个实现过程。整个进化算法的核心在与

算法模板类,如果需要实现新的算法,只需根据上文描述的数据流动情况来实现一个自

定义算法模板类,即可在 Geatpy 中运行自己设计的进化算法。

如果 Python 基础比较薄弱,可能会对上述代码中的 varTypes = [0] \* Dim 表示困惑, 这里实际上是创建了一个长度为 Dim 的、元素全为 0 的 Python list 类型列表。假如 Dim 在编写 aimFunc() 函数的过程中最容易出错的是算得的目标函数值矩阵 ObjV 和违 反约束程度矩阵 CV 不符合 Geatpy 的数据结构。在 Geatpy 中,约定 ObjV 和 CV 都是 Numpy ndarray 类型的 2D 数组,每一行对应种群的一个个体。假如是单目标优化,那么 得到的 ObjV 就应该是一个列向量;假如只有一个约束条件,那么得到的 CV 就应该是 一个列向量。

• 返回结果。 例如: main.py MyProblem.py import geatpy as ea # import geatpy from MyProblem import MyProblem # 导入自定义问题接口

构造函数完成种群对象的实例化。通过查看 Population.py 可以发现构造方法里面并没 有初始化种群对象的染色体属性,这也就对应执行脚本中的注释"此时种群还没被初始

后,这个种群才算是被"初始化"。

两点交叉算子的对象: ea.Xovdp(XOVR = 0.7)。在这一步,将跳转到 Xovdp.py 这个文件 中,完成交叉算子对象的实例化。 这里需要提及 Xovdp 这个类和 xovdp() 这个内核函数的区别:前者是在 EA 框架中

import geatpy as ea # 學入geatpy/# from sys import path as paths
from os import path
paths.append(path.split(path.split(path.realpath(\_file\_\_))[0])[0]) ss soea\_SEGA\_templet(ea.SoeaAlgorithm):

def \_\_init\_\_(self, problem, population):
 ea.SoeaAlgorithm.\_\_init\_\_(self, problem, population) # 先调用父类构造方法
 if population.ChromNum != 1:
 raise RuntimeError('传入的种群对象必须是单染色体的种群类型。')

self.name = 'SEGA'

while self.terminated(population) == False: offspring = population[ea.selecting(self.selFunc, population.FitnV, NIND)] offspring.Chrom = self.recOper.do(offspring.Chrom) # 54 offspring.Chrom = self.mutOper.do(offspring.Encoding, offspring.Chrom, offspring.Field) # %# self.call\_aimFunc(offspring) # population = population + offspring # 父子合為 population.FitnV = ea.scaling(population.ObjV, population.CV, self.problem.maxormins) # 计算适应度

population = *self*.population NIND = population.sizes *self*.initialization() # 初始化等法模模的一些动态参数 if prophetPop is not None: population = (prophetPop + population)[:NIND] # 福入先知种耕 population.FitnV = ea.scaling(population.ObjV, population.CV, self.problem.maxormins) # 计算适应度

population: All aimFunc(population) \* の別に対象を言葉に self.call aimFunc(population) # 计算序解的目标函数值 # 插入先舱知识(注意:这里不会对先知符新prophetPop的合法性进行检查,故应确保prophetPop是一个种新类且拥有合法的Chrom、ObjV、Phen等属性)

设置的一个类,目的是为了更方便地调用内核函数 xovdp()。而真正进行两点交叉的是 在内核函数 xovdp() 中。有了这个 Xovdp 类的对象,就可以实现"先实例化、设置好相 关的参数,然后再调用 xovdp()"这样的功能。 4、回到 main.py,紧接着便是调用算法模板对象的 run() 函数,这里就开始了进化 优化了。具体代码将跳转到 soea\_SEGA\_templet.py 文件的 run() 函数中。可以发现里面 的一些代码在 soea SEGA\_templet.py 文件中并没有定义,例如判断进化是否应该停止的

self.terminated()函数。如果了解Python面向对象编程,便显然可知道这个self.terminated()

实际上是 soea\_SEGA\_templet 算法模板类的父类: SoeaAlgorithm 所实现的,详见 Algo-

可能会对一些特殊的写法有些疑惑,比如: "population = population + offspring"。这个

实际上是 Population 种群类的一个运算符重载,详见 Population.py,它把加号"+"重载

5、在 soea\_SEGA\_templet.py 中可以看到这个进化算法的完整流程。这里有些用户